1 Impulse Tutorial: Generating HDL from C-Language

Overview

This Getting Started tutorial demonstrates how to compile a simple digital signal processing (DSP) filter written in C into HDL, ready for FPGA synthesis. The goal of this application will be to generate a 16-bit, 12-tap FIR filter as hardware in the form of either VHDL or Verilog. Although this is a relatively simple example in terms of the required lines of C code, it does illustrate some key concepts of Impulse C including the use of streaming and pipelining for high performance.

This tutorial covers the basics of C-to-HDL compilation, using a single C-language process. Additional tutorials extend the concepts described in this tutorial and cover desktop simulation and debugging, as well as advanced optimization techniques for increased performance.

This tutorial will require approximately 20 minutes to complete, including software run times.

Steps

For additional information about Impulse CoDeveloper, including detailed tutorials describing more advanced design techniques, please visit the Tutorials page at the following location:

www.ImpulseAccelerated.com/Tutorials

1.1 Loading the FIR12 Filter Application

FIR Filter Tutorial, Step 1

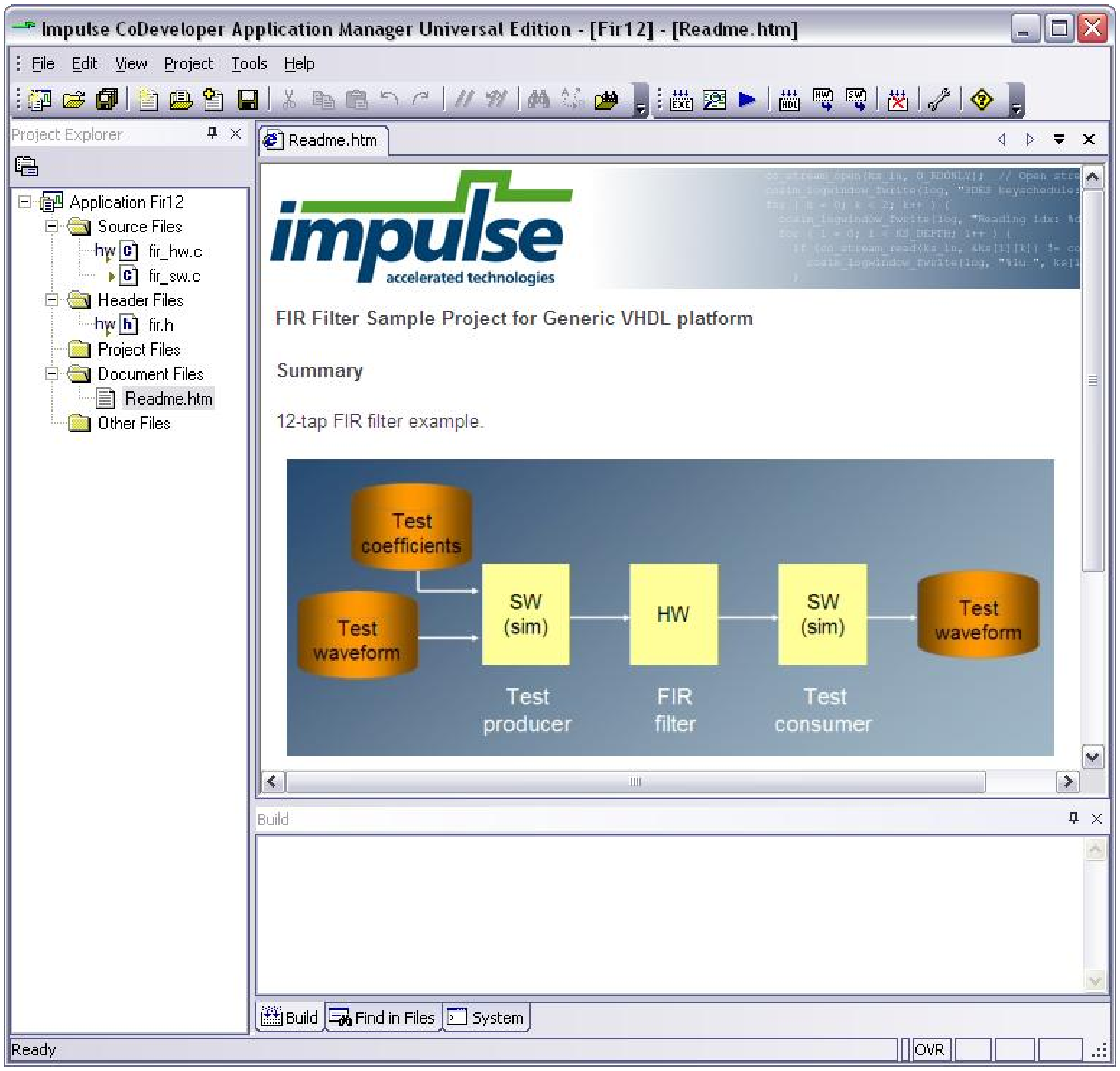
To begin, start the **CoDeveloper** Application Manager:

Start -> **Programs** -> **Impulse Accelerated Technologies** -> **CoDeveloper -> CoDeveloper Application Manager**

Open the **FIR51** sample project by selecting **Open Project** from the **File** menu, or by clicking the **Open Project** toolbar button. Navigate to the **.\Examples\DSP\Fir12\** directory within your CoDeveloper installation. (You may wish to copy this example to an alternate directory before beginning.)

The project file is also available from the CoDeveloper Start Page, in the **Help and Support** tab.

After loading the project, you will see a **Readme** file with a block diagram, and a **Project Explorer** window as shown below:



FIR Filter Sample Project for Generic VHDL platform

Summary

12-tap FIR filter example.

Source files included in the **Fir12** project include:

- **Fir_hw.c** - This source file includes the C-language description of the 16-bit, 12-tap FIR filter, including its I/O.
- **Fir_sw.c** - This source file includes a set of software testing routines including a **main()** function, and **consumer** and **producer** software processes as illustrated in the block diagram.
- **fir.h** - This source file includes common declarations used in both the FIR filter description, and in the test routines.

You can open any of these three files by simply double-clicking on the file name in the Project Explorer window. In the next step, we will describe in detail how this example works.

Next Step

Understanding the FIR12 Application

1.2 Understanding the FIR12 Application

FIR Filter Tutorial, Step 2

Before compiling the FIR application to create hardware, let's first take a moment to understand its basic operation.

The FIR Filter C-Language Process

The specific process that we will be compiling to hardware is represented by the following function, which is located in **Fir_hw.c**:

```
void fir(co_stream filter_in, co_stream filter_out)
```

This C-language subroutine represents an *Impulse C process*. A *process* in Impulse C is a module of code, expressed as a **void** subroutine, that describes a hardware or software component.

If you are an experienced hardware designer, you can simply think of a process as being analogous to a VHDL *entity*, or to a Verilog *module*.

If you are a software programmer, you can think of a process as being a subroutine that will loop forever, in a seperate *thread of execution* from other processes.

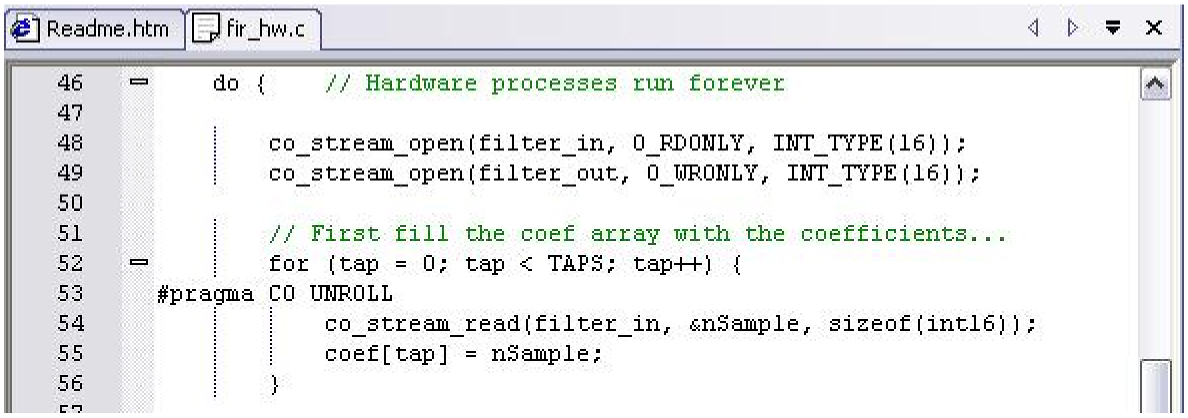
Our **fir** function has no return value, and has two interfaces that have been defined using Impulse C **co_stream** data types. These two *streams* are used to:

- Read in a set of 12 filter coefficients, and then a stream of sample data on the **filter_in** stream.
- Write out the filter values on the **filter_out** stream.

If you are a hardware designer, you can think of a **co_stream** as being a representation of a first-in, first-out (FIFO) buffer.

If you are a software programmer, you can think of a **co_stream** is being roughly analogous to a FILE type in C. Rather than reading and writing files on a disk, however, we will use the **co_stream** type to transfer data between multiple *parallel processes*.

Scroll down in the source code to view the algorithm and its nested loops:



```
    do {    // Hardware processes run forever

        co_stream_open(filter_in, O_RDONLY, INT_TYPE(16));
        co_stream_open(filter_out, O_WRONLY, INT_TYPE(16));

        // First fill the coef array with the coefficients...
        for (tap = 0; tap < TAPS; tap++) {
#pragma CO UNROLL
            co_stream_read(filter_in, &nSample, sizeof(int16));
            coef[tap] = nSample;
        }
```

Notice that the subroutine includes an outer **do-while(1)** loop, indicating that the subroutine will execute endlessly. This subroutine describes a persistent, always-running process.

Within this loop, observe how the **co_stream_open**, **co_stream_read**, **co_stream_write** and

co_stream_close functions are used to manage the movement of data through the filter. These functions provide you, the C programmer, with a concise and platform-portable way to express streaming data. Impulse C supports a number of similar functions that can be used to describe the movement and management of process-to-process data.

The **fir** function begins by reading 12 coefficients from the **filter_in** stream and storing the resulting data into a local array (**coef**). The function then reads and begins processing the data inputs, one sample at a time. Results of filtering are written to the output stream **filter_out**.

If you scroll down further in the algorithm description, you will find a **while** loop that describes the actual filtering operation, which is an iterative multiply-accumulate operation as shown below:

```
// Read values from the stream
while ( co_stream_read(filter_in, &nSample, sizeof(int16)) == co_err_none ) {
#pragma CO PIPELINE
#pragma CO SET StageDelay 100

 IF_SIM(samplesread++;)
 firbuffer[TAPS-1] = nSample;

 accum = 0;
 for (tap = 0; tap < TAPS; tap++) {
 #pragma CO UNROLL
    accum += firbuffer[tap] * coef[tap];
 }
 nFiltered = accum >> 2;
 co_stream_write(filter_out, &nFiltered, sizeof(int16));
 IF_SIM(sampleswritten++;)

 for (tap = 1; tap < TAPS; tap++) {
 #pragma CO UNROLL
    firbuffer[tap-1] = firbuffer[tap];
 }
}
```

This loop includes two inner loops and a simple set of calculations to iterate over every 12-sample segment of the incoming data to perform the filtering operation. In each iteration of the **while** loop, filtered data is written to the output stream using **co_stream_write**.

The above loop illustrates a very common pattern for describing filters using Impulse C: a C-language loop iterates on the incoming data, some processing occurs on that data, and results are written to the outputs using streaming (as shown here) or other methods.

You probably noticed the use of three pragmas in the code (**PIPELINE**, **UNROLL** and **SET StageDelay**). These pragmas are the subject of a more detailed tutorial on optimization techniques, but to summarize (in the order these pragmas are used in the above code):

- The **CO PIPELINE** pragma indicates that we want the **while** loop to be implemented as a hardware pipeline for high throughput. If the hardware compiler is able to generate a *perfect pipeline* with a *rate of 1*, then we can expect this loop to iterate in hardware as fast as *one sample per clock cycle*, even if the computations within the loop require more than one cycle.
- The **CO SET** pragma allows us to specify certain characteristics for the generated hardware. In this case we are setting a **StageDelay** constraint that instructs the optimizer to limit the combinational logic depth of any pipeline stage. If any generated pipeline stage exceeds this constraint, the optimizer will add additional pipeline stages to better balance the pipeline and allow the hardware to operate at a high clock rate.
- The **UNROLL** pragma instructs the optimizer to remove (by unrolling) a loop so that all iterations of that loop operate in parallel. Unrolling requires that the loop obey certain rules (such as having a fixed number of loop iterations) but can have dramatic impacts on performance, at the expense of

additional FPGA logic being generated.

The FIR Filter Configuration Subroutine

The **fir** subroutine described above represents the algorithm to be implemented as hardware in the FPGA. To complete the application, however, we need to include one additional routine that describes the I/O connections and other compile-time characteristics for this application. This *configuration routine* serves three important purposes, allowing us to:

1. define I/O characteristics such as FIFO depths and the sizes of shared memories.
2. instantiate and interconnect one or more copies of our Impulse C process.
3. optionally assign physical, chip-level names and/or locations to specific I/O ports.

This example only includes one hardware process (the FIR filter) but it also includes the two testing routines that we described earlier, **producer** and **consumer**. Our configuration routine therefore includes statements that describe how the **producer**, **fir** and **consumer** processes are connected together. The complete configuration routine is shown below:

```
void config_fir(void *arg)
{
 co_stream waveform_in;
 co_stream waveform_out;
 co_process fir_process;
 co_process producer_process;
 co_process consumer_process;
 IF_SIM(cosim_logwindow_init();)

 waveform_in = co_stream_create("waveform_in", INT_TYPE(16), BUFSIZE);
 waveform_out = co_stream_create("waveform_out", INT_TYPE(16), BUFSIZE);

 producer_process = co_process_create("producer_process",
                                      (co_function)test_producer,
                                      1, waveform_in);
 fir_process = co_process_create("filter_process", (co_function)fir,
                                  2, waveform_in, waveform_out);
 consumer_process = co_process_create("consumer_process",
                                      (co_function)test_consumer,
                                      1, waveform_out);
 // Assign fir process to hardware elements
 co_process_config(fir_process, co_loc, "PE0");
}
```

To summarize, the **fir** subroutine describes the algorithm to be generated as FPGA hardware, while the **producer** and **consumer** subroutines (described elsewhere, in **fir_sw.c**) are used for testing purposes. The configuration routine is used to describe how these three processes communicate, and to describe other characteristics of the process I/O.

Next Step

Compiling the C Code to Create HDL

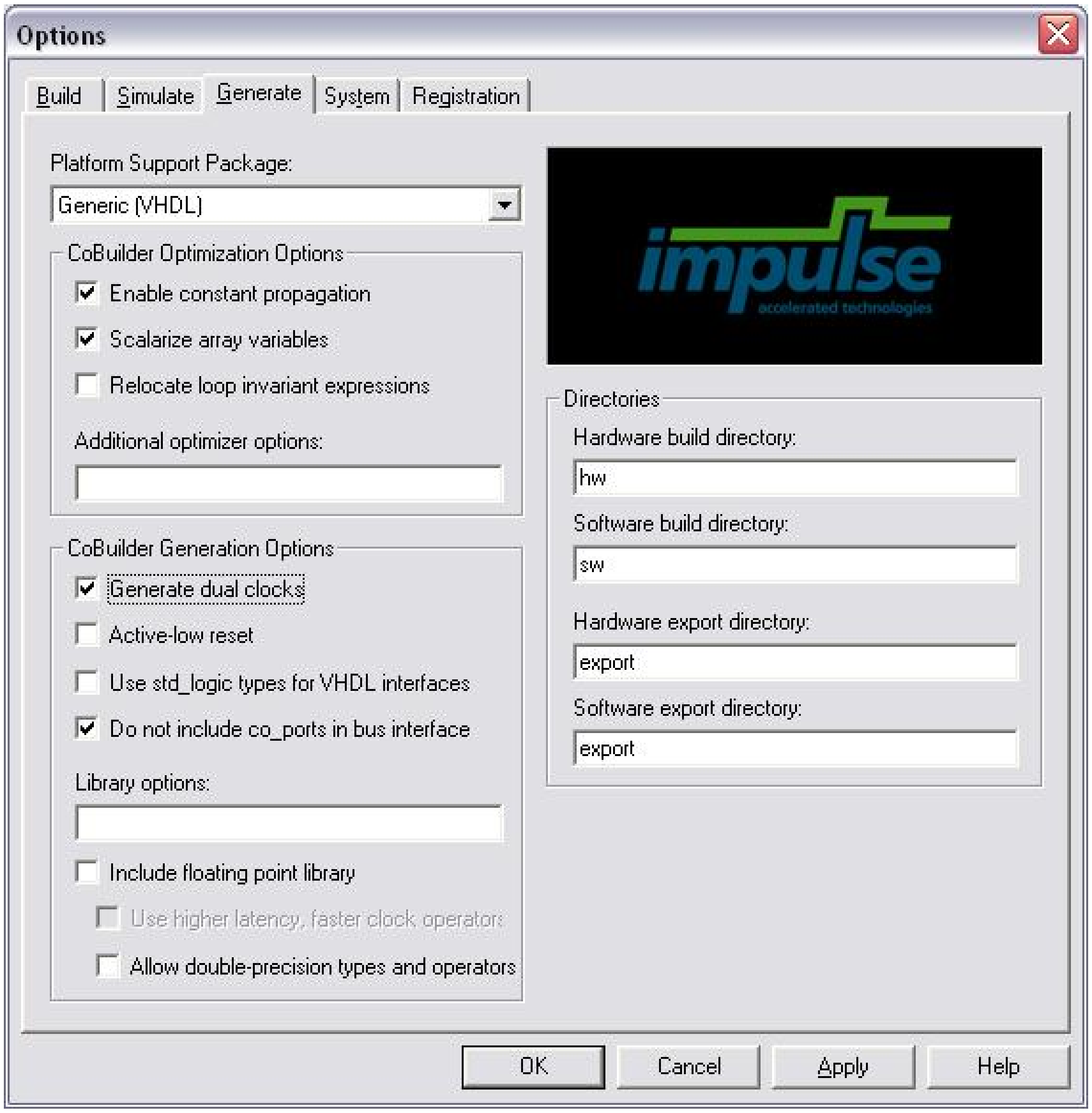
1.3 Compiling the C Code to Create HDL

FIR Filter Tutorial, Step 3

Now that you have examined the FIR filter sample code, the next step is to create FPGA hardware and related files from the C code found in the **Fir_hw.c** source file. This requires that we select a platform target, specify any needed options, and initiate the hardware compilation process.

Specifying the Platform Support Package

To specify a platform target, select **Project** -> **Options**, the select the **Generate** tab to open the **Generate Options** dialog as shown below:



Platform Support Package: Generic (VHDL)

CoBuilder Optimization Options
- [✓] Enable constant propagation
- [✓] Scalarize array variables
- [ ] Relocate loop invariant expressions

Additional optimizer options:

CoBuilder Generation Options
- [✓] Generate dual clocks
- [ ] Active-low reset
- [ ] Use std_logic types for VHDL interfaces
- [✓] Do not include co_ports in bus interface

Library options:

- [ ] Include floating point library
  - [ ] Use higher latency, faster clock operators
  - [ ] Allow double-precision types and operators

Directories
- Hardware build directory: hw
- Software build directory: sw
- Hardware export directory: export
- Software export directory: export

This dialog allows you to set various options for hardware generation, and to select a target platform. Notice that the Platform Support Package setting indicates we want to generate "Generic (VHDL)" for our output. This indicates that we have not selected a specific FPGA platform. You can click on the drop-down Platform Support Package selection list to see what kind of *platform support packages* are installed on your system. We will use the default setting for this sample hardware generation.

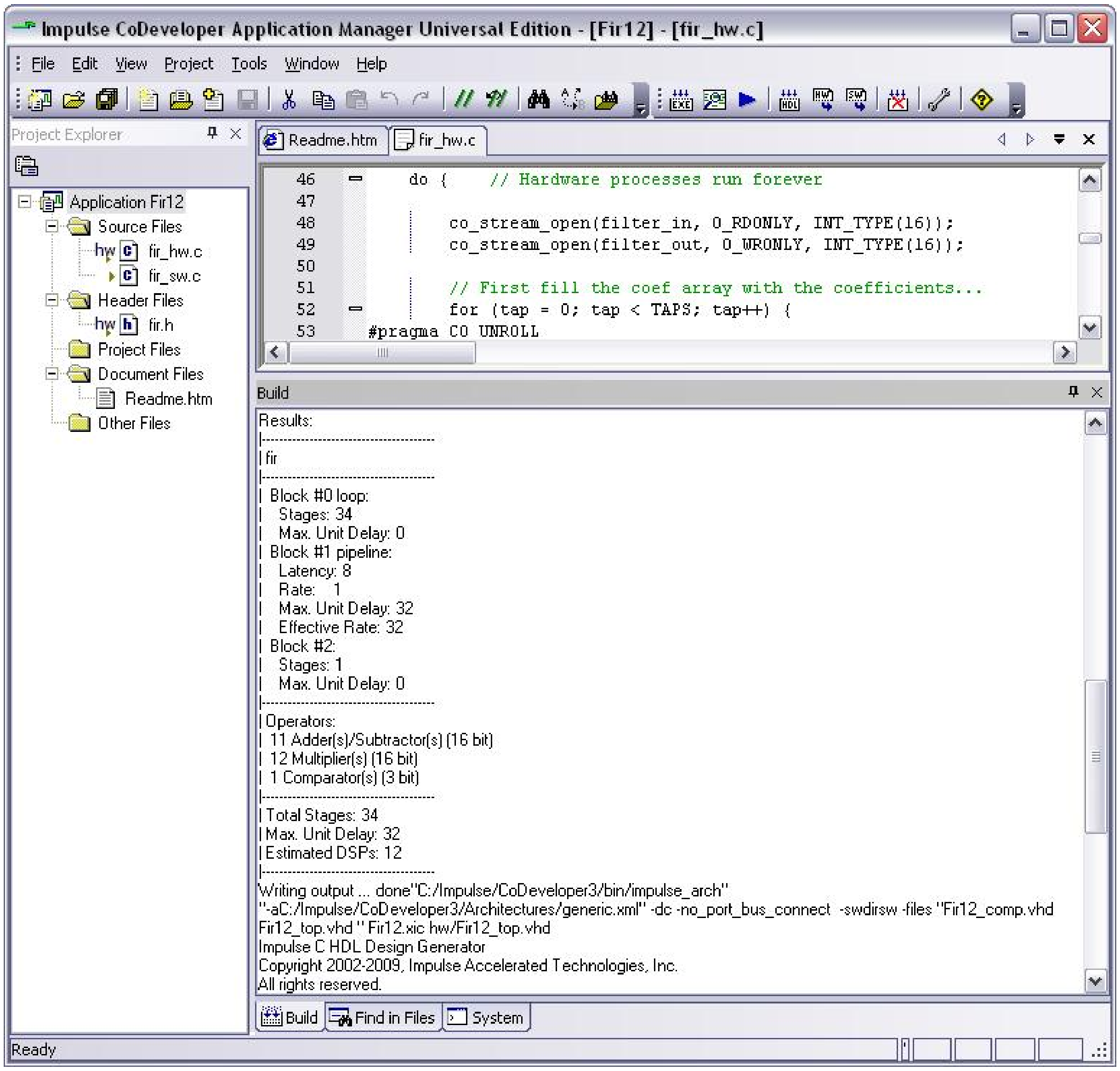
Note: if you would prefer to generate Verilog, you can change the setting to "Generic (Verilog)" before continuing.

Other options on this dialog allow you to set the target directory for generating and exporting your HDL, and set options related to the clock and reset hardware, and include optional hardware libraries.

Click **OK** to save the options and exit the dialog.

Generating HDL for the Hardware Process

To generate hardware in the form of HDL files, select **Project** -> **Generate HDL**. A series of processing steps will run in the **Build** console window as shown below (you can use your mouse to expand the **Build** window as shown):

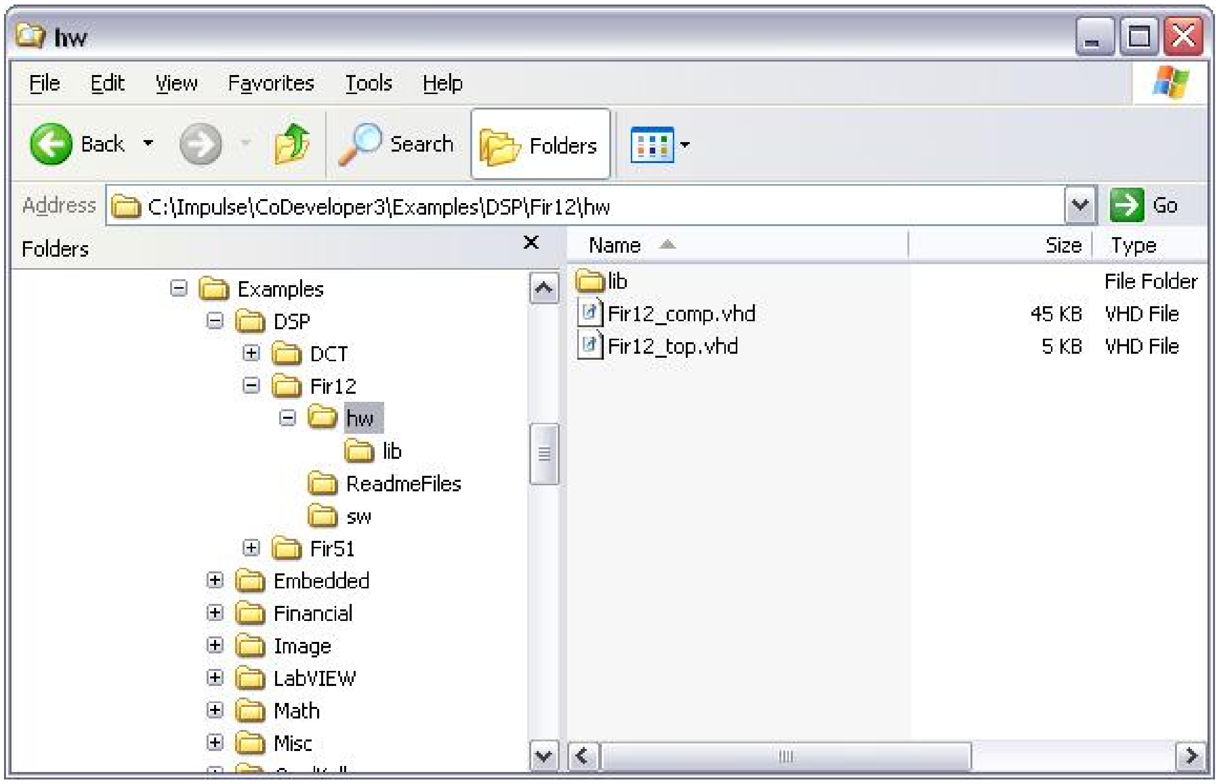


```
    do {    // Hardware processes run forever

        co_stream_open(filter_in, O_RDONLY, INT_TYPE(16));
        co_stream_open(filter_out, O_WRONLY, INT_TYPE(16));

        // First fill the coef array with the coefficients...
        for (tap = 0; tap < TAPS; tap++) {
#pragma CO UNROLL
```

```
Results:
|----------------------------------------
| fir
|----------------------------------------
| Block #0 loop:
|   Stages: 34
|   Max. Unit Delay: 0
| Block #1 pipeline:
|   Latency: 8
|   Rate:    1
|   Max. Unit Delay: 32
|   Effective Rate: 32
| Block #2:
|   Stages: 1
|   Max. Unit Delay: 0
|----------------------------------------
| Operators:
|  11 Adder(s)/Subtractor(s) (16 bit)
|  12 Multiplier(s) (16 bit)
|  1 Comparator(s) (3 bit)
|----------------------------------------
| Total Stages: 34
| Max. Unit Delay: 32
| Estimated DSPs: 12
|----------------------------------------
Writing output ... done"C:/Impulse/CoDeveloper3/bin/impulse_arch"
"-aC:/Impulse/CoDeveloper3/Architectures/generic.xml" -dc -no_port_bus_connect  -swdirsw -files "Fir12_comp.vhd
Fir12_top.vhd " Fir12.xic hw/Fir12_top.vhd
Impulse C HDL Design Generator
Copyright 2002-2009, Impulse Accelerated Technologies, Inc.
All rights reserved.
```

The messages generated by the hardware compiler include estimates of loop latencies and pipeline rates as well as estimates of the number of required hardware operations, as shown above. These messages can help you to quickly evaluate the effectiveness of your C-language coding methods, allowing you to iteratively refactor and improve your algorithms *before* going through a potentially long process of FPGA synthesis.

When the optimization and C-to-HDL processing has completed you will have two resulting HDL files (either VHDL or Verilog) created in the **hw** subdirectory of your project directory, including a **lib** subdirectory, as shown below:



Name	Size	Type
lib		File Folder
Fir12_comp.vhd	45 KB	VHD File
Fir12_top.vhd	5 KB	VHD File

Next Step

Examining the Generated HDL

1.4 Examining the Generated HDL

FIR Filter Tutorial, Step 4

You have successfully generated HDL from a C-language description. Let's take a moment to examine the generated HDL and see how it relates to the original C code. We will examine the generated hardware in the form of VHDL; if you generated Verilog, the syntax will be different but the generated hardware will be similar.

Top-Level HDL Entity (Module)

Recall that in our original C code, the I/O interfaces for the **fir** process were described using **co_stream** data, and using stream-related functions such as **co_stream_read** and **co_stream_write**.

In the generated hardware, the HDL file with the **_top** file name suffix (in this case **Fir12_top.vhd**) represents the top-level I/O implementing these streaming interfaces, as shown below:

```
entity fir_arch is
  port (
    reset : in std_ulogic;
    sclk : in std_ulogic;
    clk : in std_ulogic;
    p_producer_process_waveform_in_en : in std_ulogic;
    p_producer_process_waveform_in_eos : in std_ulogic;
    p_producer_process_waveform_in_data : in std_ulogic_vector (15 downto 0);
    p_producer_process_waveform_in_rdy : out std_ulogic;
    p_consumer_process_waveform_out_en : in std_ulogic;
    p_consumer_process_waveform_out_data : out std_ulogic_vector (15 downto 0);
    p_consumer_process_waveform_out_eos : out std_ulogic;
    p_consumer_process_waveform_out_rdy : out std_ulogic);
end;
```

For each of the two streams, notice that there are data and flow control signals with the suffix **_data**, **_en**, **_rdy** and **_eos**. These flow control hardware signals are documented in the Impulse User's Guide and can be used to connect other streaming hardware (as as analog-to-digital inputs, video inputs and other streaming hardware) directly to an Impulse-generated streaming hardware process.

Also notice the names used when generating the I/O signals. Because we did not specify actual port names for our input and output streams, the compiler has assigned names to the hardware streams based on their source and destination, in this case the **producer** and **consumer** processes. In a real-world application we might choose to assign specific names to these streams, using a **co_port_create** function, or choose a platform support package that automatically generates appropriately named I/O wrappers for our target platform.

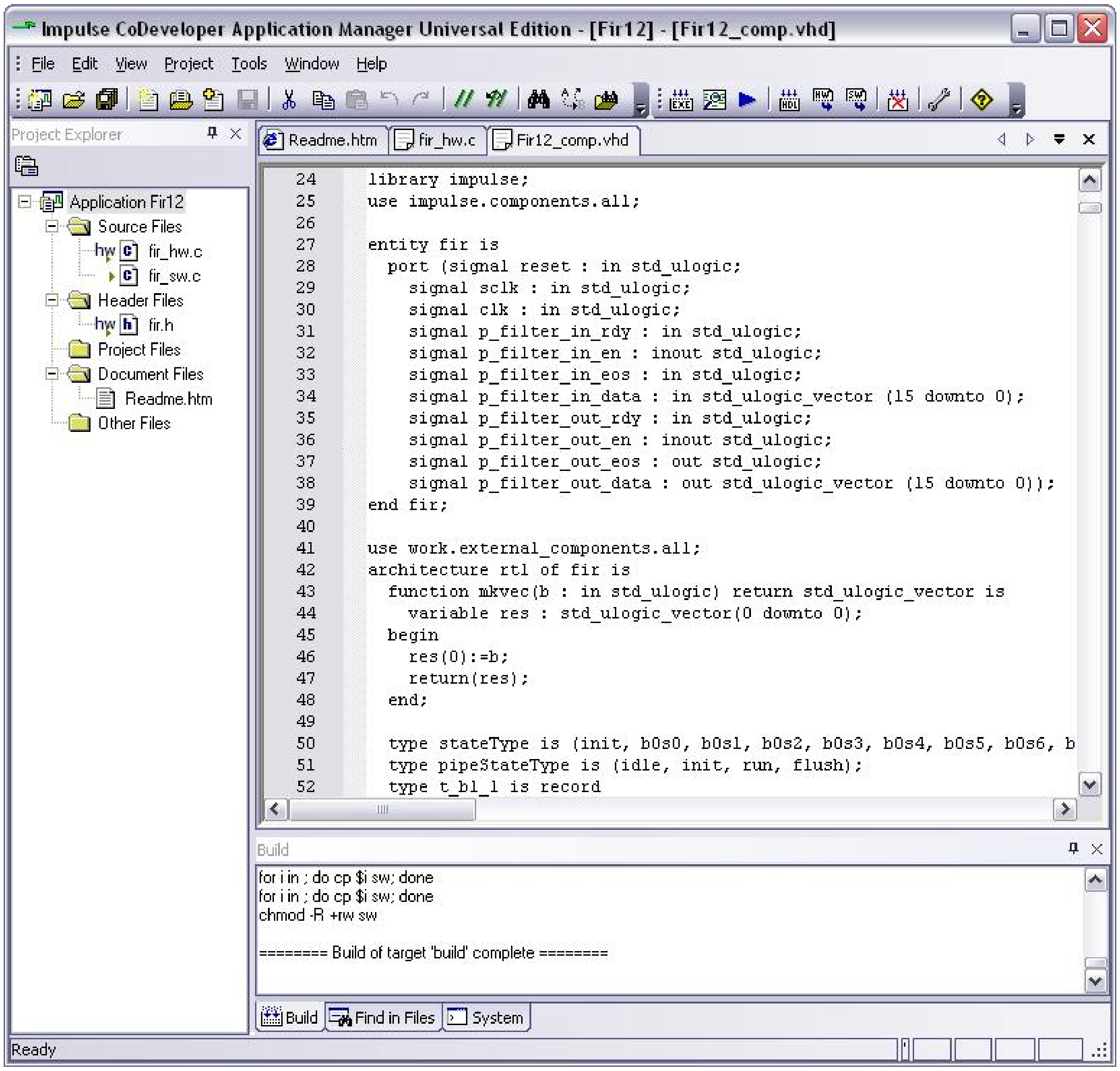
Moving down in the **Fir12_top.vhd** file, we can find the following component instantiations (port maps have been removed for brevity):

```
  filter_process: fir
    port map (
    );

  inst0: stream_dc
    generic map (
      datawidth => 16,
      addrwidth => 1
    )
    port map (
    );

  inst1: stream_dc
    generic map (
      datawidth => 16,
      addrwidth => 1
    )
    port map (
    );
```

The **filter_process** component is an instantiation of our **fir** subroutine, as specified in our C-code using the **co_process_create** function. The **inst0** and **inst1** components are instantiations of Impulse stream components (FIFO elements), as specified using the **co_stream_create** function.

Component-Level HDL Entity (Module)

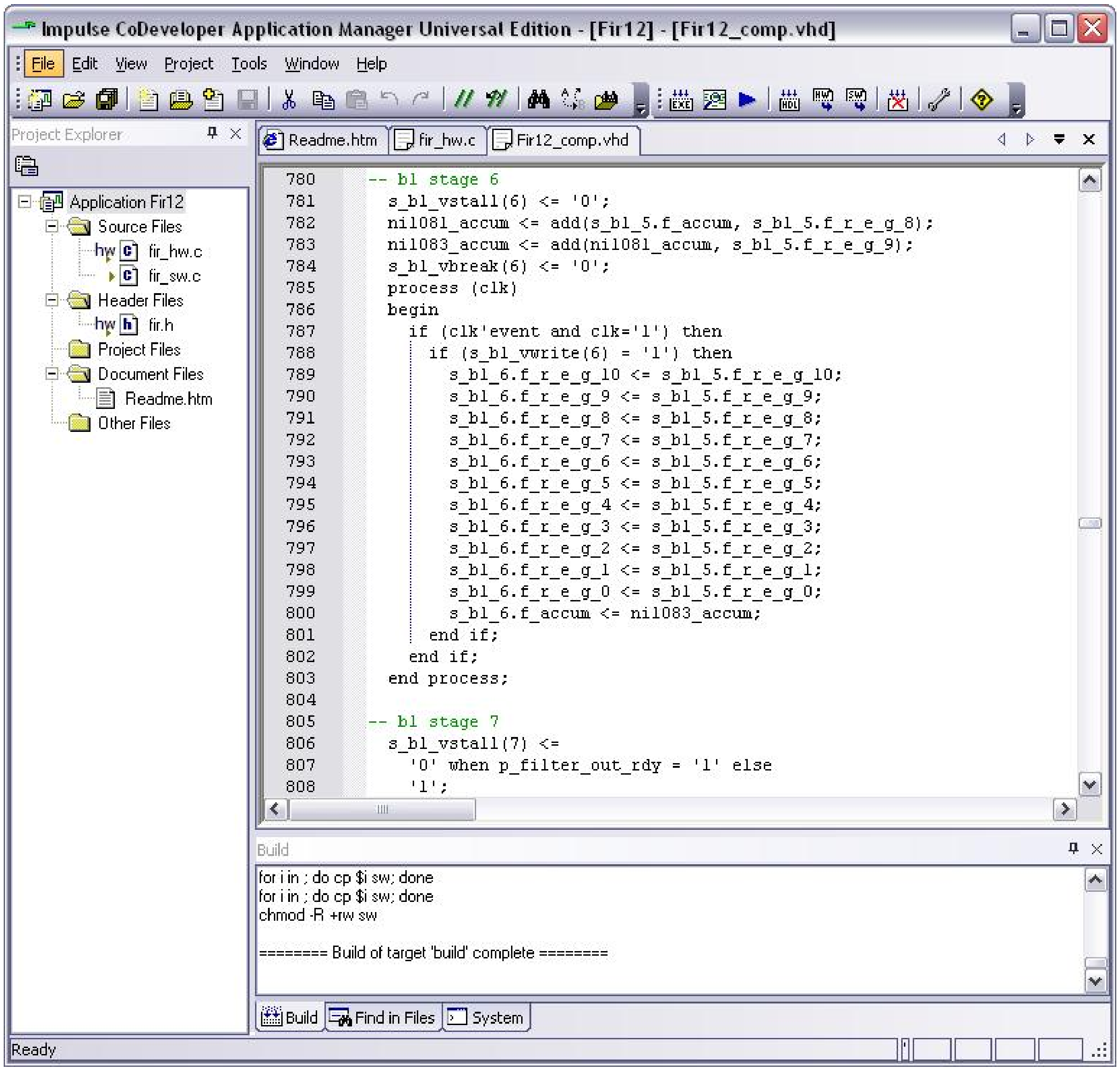
To view the lower-level HDL code for the **fir** subroutine, open the **Fir12_comp.vhd** file as shown below:



```
library impulse;
use impulse.components.all;

entity fir is
  port (signal reset : in std_ulogic;
    signal sclk : in std_ulogic;
    signal clk : in std_ulogic;
    signal p_filter_in_rdy : in std_ulogic;
    signal p_filter_in_en : inout std_ulogic;
    signal p_filter_in_eos : in std_ulogic;
    signal p_filter_in_data : in std_ulogic_vector (15 downto 0);
    signal p_filter_out_rdy : in std_ulogic;
    signal p_filter_out_en : inout std_ulogic;
    signal p_filter_out_eos : out std_ulogic;
    signal p_filter_out_data : out std_ulogic_vector (15 downto 0));
end fir;

use work.external_components.all;
architecture rtl of fir is
  function mkvec(b : in std_ulogic) return std_ulogic_vector is
    variable res : std_ulogic_vector(0 downto 0);
  begin
    res(0):=b;
    return(res);
  end;

  type stateType is (init, b0s0, b0s1, b0s2, b0s3, b0s4, b0s5, b0s6, b
  type pipeStateType is (idle, init, run, flush);
  type t_bl_l is record
```

```
for i in ; do cp $i sw; done
for i in ; do cp $i sw; done
chmod -R +rw sw

======== Build of target 'build' complete ========
```

This HDL file includes the state machines and other logic that implements the parellized and pipelined operations described in C. This example includes a pipelined inner code loop with an unrolled loop, which results in a substantial amount of HDL code being generated:



```
-- bl stage 6
  s_bl_vstall(6) <= '0';
  ni1081_accum <= add(s_bl_5.f_accum, s_bl_5.f_r_e_g_8);
  ni1083_accum <= add(ni1081_accum, s_bl_5.f_r_e_g_9);
  s_bl_vbreak(6) <= '0';
  process (clk)
  begin
    if (clk'event and clk='1') then
      if (s_bl_vwrite(6) = '1') then
        s_bl_6.f_r_e_g_10 <= s_bl_5.f_r_e_g_10;
        s_bl_6.f_r_e_g_9 <= s_bl_5.f_r_e_g_9;
        s_bl_6.f_r_e_g_8 <= s_bl_5.f_r_e_g_8;
        s_bl_6.f_r_e_g_7 <= s_bl_5.f_r_e_g_7;
        s_bl_6.f_r_e_g_6 <= s_bl_5.f_r_e_g_6;
        s_bl_6.f_r_e_g_5 <= s_bl_5.f_r_e_g_5;
        s_bl_6.f_r_e_g_4 <= s_bl_5.f_r_e_g_4;
        s_bl_6.f_r_e_g_3 <= s_bl_5.f_r_e_g_3;
        s_bl_6.f_r_e_g_2 <= s_bl_5.f_r_e_g_2;
        s_bl_6.f_r_e_g_1 <= s_bl_5.f_r_e_g_1;
        s_bl_6.f_r_e_g_0 <= s_bl_5.f_r_e_g_0;
        s_bl_6.f_accum <= ni1083_accum;
      end if;
    end if;
  end process;

-- bl stage 7
  s_bl_vstall(7) <=
    '0' when p_filter_out_rdy = '1' else
    '1';
```

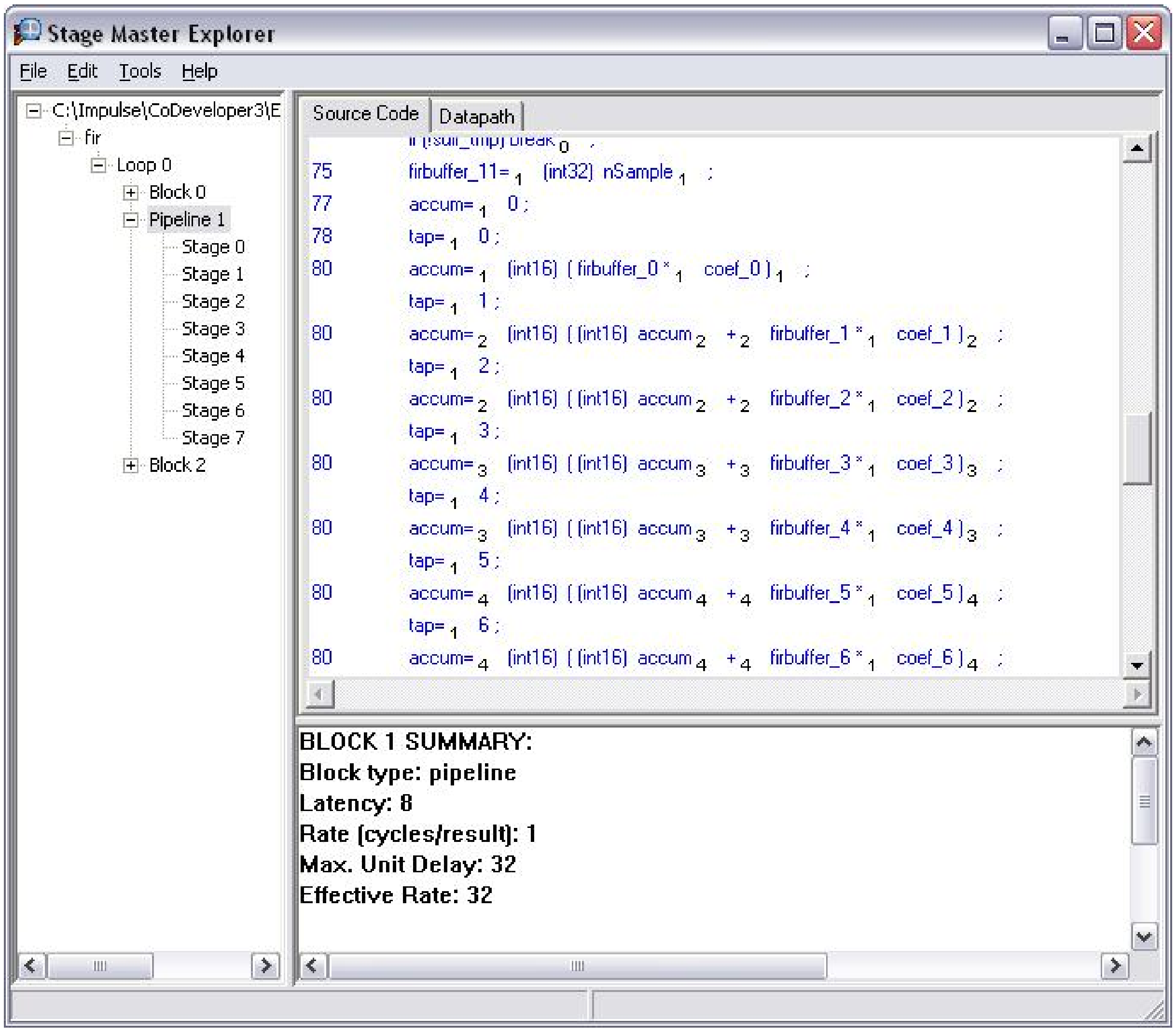
```
for i in ; do cp $i sw; done
for i in ; do cp $i sw; done
chmod -R +rw sw

======== Build of target 'build' complete ========
```

When you examine this generated HDL code, keep in mind that the number of lines of HDL code is not directly related to the size of the FPGA resources. In this case, because of the loop unrolling and pipelining, a large number of intermediate signals are generated by the compiler. These intermediate signals are optimized away by the FPGA synthesis tool, resulting far less logic than the lines of HDL code might indicate.

Note: the amount of FPGA resources and final performance for such a filter will depend on the selected FPGA platform, on the synthesis settings, and on what other hardware elements are being combined with this filter in the complete system. In the case of this algorithm (a 16-bit, fully pipelined and parallelized 12-tap filter), you can expect to use approximately 12 DSP slices in a typical FPGA device.

Viewing the Results Graphically

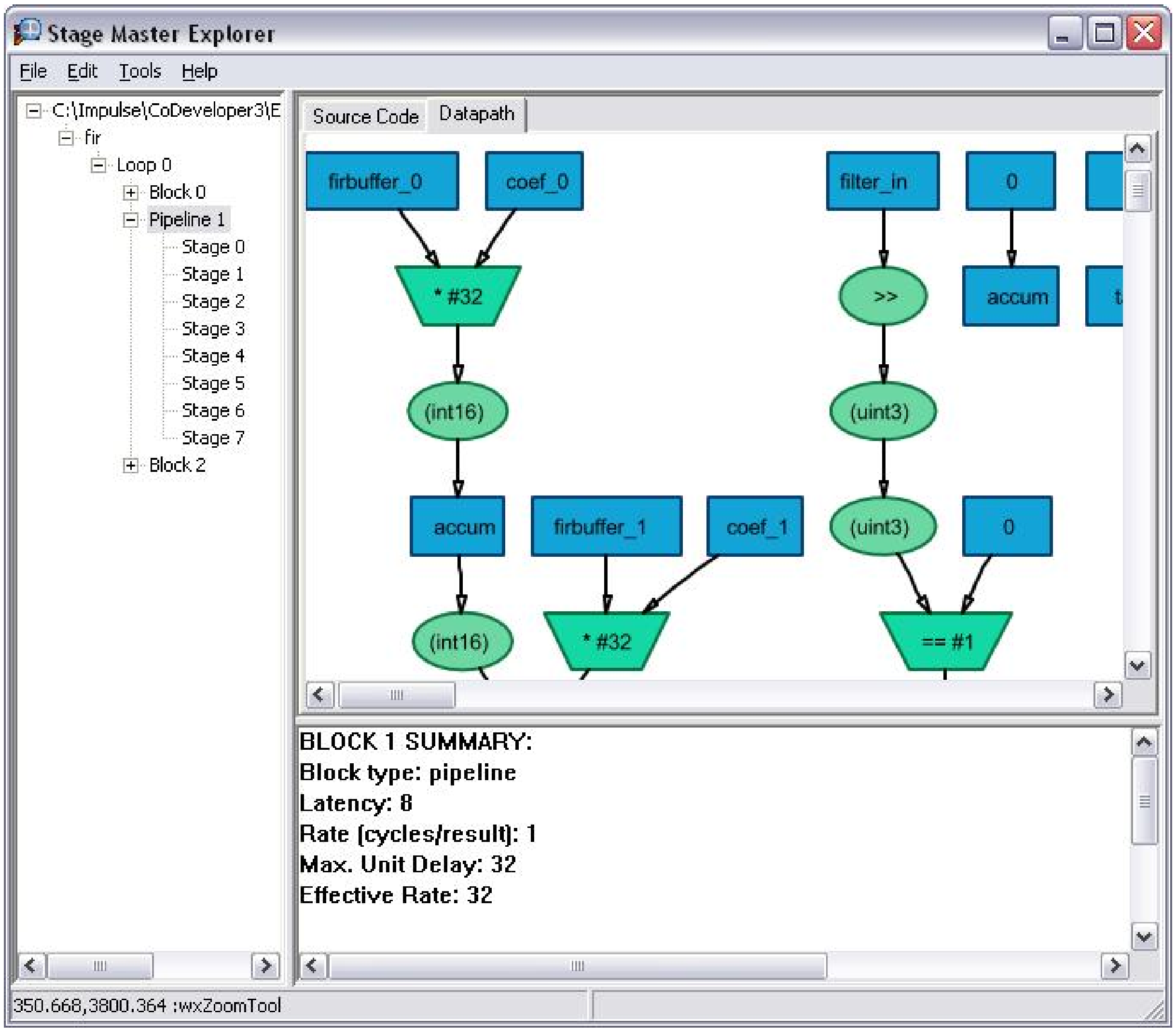
To see a graphical representation of the generated hardware, invoke the Stage Master Explorer tool by selecting **PTools -> State Master Explorer**. When prompted, select the **Fir51.xic** compiler-intermediate file. You can use Stage Master Explorer to view an expanded form of the original source code, and to view a graph of the unrolled and pipelined inner loop as shown below:



```
C:\Impulse\CoDeveloper3\E
  fir
    Loop 0
      Block 0
      Pipeline 1
        Stage 0
        Stage 1
        Stage 2
        Stage 3
        Stage 4
        Stage 5
        Stage 6
        Stage 7
      Block 2
```

```
if (!still_tmp) break 0 ;
firbuffer_11= 1  (int32) nSample 1  ;
accum= 1  0 ;
tap= 1  0 ;
accum= 1  (int16) ( firbuffer_0 * 1  coef_0 ) 1  ;
tap= 1  1 ;
accum= 2  (int16) ( (int16) accum 2  + 2  firbuffer_1 * 1  coef_1 ) 2  ;
tap= 1  2 ;
accum= 2  (int16) ( (int16) accum 2  + 2  firbuffer_2 * 1  coef_2 ) 2  ;
tap= 1  3 ;
accum= 3  (int16) ( (int16) accum 3  + 3  firbuffer_3 * 1  coef_3 ) 3  ;
tap= 1  4 ;
accum= 3  (int16) ( (int16) accum 3  + 3  firbuffer_4 * 1  coef_4 ) 3  ;
tap= 1  5 ;
accum= 4  (int16) ( (int16) accum 4  + 4  firbuffer_5 * 1  coef_5 ) 4  ;
tap= 1  6 ;
accum= 4  (int16) ( (int16) accum 4  + 4  firbuffer_6 * 1  coef_6 ) 4  ;
```

```
BLOCK 1 SUMMARY:
Block type: pipeline
Latency: 8
Rate (cycles/result): 1
Max. Unit Delay: 32
Effective Rate: 32
```



BLOCK 1 SUMMARY:
Block type: pipeline
Latency: 8
Rate (cycles/result): 1
Max. Unit Delay: 32
Effective Rate: 32

To view the graph as shown above, use your mouse to click and drag a section of the displayed pipeline graph.

Next Steps

You have now completed this tutorial. At this point you may want to explore other examples provided with CoDeveloper, or explore some of the more advanced, platform-specific tutorials to learn more about how to use the generated HDL in actual hardware.

For additional information other detailed tutorials, please visit the Tutorials page at the following location:

www.ImpulseAccelerated.com/Tutorials